Card Tricks

Welcome to Card Tricks on Exercism's Go Track. If you need help running the tests or submitting your code, check out `HELP.md`. If you get stuck on the exercise, check out `HINTS.md`, but try and solve it without using those first :)

Introduction

Slices

Slices in Go are similar to lists or arrays in other languages. They hold several elements of a specific type (or interface).

Slices in Go are based on arrays. Arrays have a fixed size. A slice, on the other hand, is a dynamically-sized, flexible view of the elements of an array.

A slice is written like `[]T` with `T` being the type of the elements in the slice:

```
var empty []int                 // an empty slice
withData := []int{0,1,2,3,4,5}  // a slice pre-filled with some data
```

You can get or set an element at a given zero-based index using the square-bracket notation:

```
withData[1] = 5
x := withData[1] // x is now 5
```

You can create a new slice from an existing slice by getting a range of elements. Once again using square-bracket notation, but specifying both a starting (inclusive) and ending (exclusive) index. If you don't specify a starting index, it defaults to 0. If you don't specify an ending index, it defaults to the length of the slice.

```
newSlice := withData[2:4]
// => []int{2,3}
newSlice := withData[:2]
```

```
// => []int{0,1}
newSlice := withData[2:]
// => []int{2,3,4,5}
newSlice := withData[:]
// => []int{0,1,2,3,4,5}
```

You can add elements to a slice using the `append` function. Below we append `4` and `2` to the `a` slice.

```
a := []int{1, 3}
a = append(a, 4, 2)
// => []int{1,3,4,2}
```

`append` always returns a new slice, and when we just want to append elements to an existing slice, it's common to reassign it back to the slice variable we pass as the first argument as we did above.

`append` can also be used to merge two slices:

```
nextSlice := []int{100,101,102}
newSlice  := append(withData, nextSlice...)
// => []int{0,1,2,3,4,5,100,101,102}
```

Variadic Functions

Usually, functions in Go accept only a fixed number of arguments. However, it is also possible to write variadic functions in Go.

A variadic function is a function that accepts a variable number of arguments.

If the type of the last parameter in a function definition is prefixed by ellipsis `...`, then the function can accept any number of arguments for that parameter.

```
func find(a int, b ...int) {
    // ...
}
```

In the above function, parameter `b` is variadic and we can pass 0 or more arguments to `b`.

```
find(5, 6)
find(5, 6, 7)
find(5)
```

```
The variadic parameter must be the last parameter of the function.
```

The way variadic functions work is by converting the variable number of arguments to a slice of the type of the variadic parameter.

Here is an example of an implementation of a variadic function.

```
func find(num int, nums ...int) {
    fmt.Printf("type of nums is %T\n", nums)

    for i, v := range nums {
        if v == num {
            fmt.Println(num, "found at index", i, "in", nums)
            return
        }
    }

    fmt.Println(num, "not found in ", nums)
}

func main() {
    find(89, 90, 91, 95)
    // =>
    // type of nums is []int
    // 89 not found in  [90 91 95]

    find(45, 56, 67, 45, 90, 109)
    // =>
    // type of nums is []int
    // 45 found at index 2 in [56 67 45 90 109]

    find(87)
    // =>
    // type of nums is []int
    // 87 not found in  []
}
```

In line `find(89, 90, 91, 95)` of the program above, the variable number of arguments to the find function are `90` , `91` and `95` . The `find` function expects a variadic int parameter after

`num` . Hence these three arguments will be converted by the compiler to a slice of type `int []int{90, 91, 95}` and then it will be passed to the find function as `nums` .

Sometimes you already have a slice and want to pass that to a variadic function. This can be achieved by passing the slice followed by `...` . That will tell the compiler to use the slice as is inside the variadic function. The step described above where a slice is created will simply be omitted in this case.

```
list := []int{1, 2, 3}
find(1, list...) // "find" defined as shown above
```

Instructions

As a magician-to-be, Elyse needs to practice some basics. She has a stack of cards that she wants to manipulate.

To make things a bit easier she only uses the cards 1 to 10.

1. Create a slice with certain cards

When practicing with her cards, Elyse likes to start with her favorite three cards of the deck: 2, 6 and 9. Write a function `FavoriteCards` that returns a slice with those cards in that order.

```
cards := FavoriteCards()
fmt.Println(cards)
// Output: [2 6 9]
```

2. Retrieve a card from a stack

Return the card at position `index` from the given stack.

```
card := GetItem([]int{1, 2, 4, 1}, 2) // card == 4
```

If the index is out of bounds (ie. if it is negative or after the end of the stack), we want to return `-1` :

```
card := GetItem([]int{1, 2, 4, 1}, 10) // card == -1
```

```
By convention in Go, an error is returned instead of returning an "out-of-band" value.
Here the "out-of-band" value is `-1` when a positive integer is expected.
When returning an error, it's considered idiomatic to return the [`zero value`](https://v
Returning an error with the proper return value will be covered in a future exercise.
```

3. Exchange a card in the stack

Exchange the card at position `index` with the new card provided and return the adjusted stack. Note that this will modify the input slice which is the expected behavior.

```
index := 2
newCard := 6
cards := SetItem([]int{1, 2, 4, 1}, index, newCard)
fmt.Println(cards)
// Output: [1 2 6 1]
```

If the index is out of bounds (ie. if it is negative or after the end of the stack), we want to append the new card to the end of the stack:

```
index := -1
newCard := 6
cards := SetItem([]int{1, 2, 4, 1}, index, newCard)
fmt.Println(cards)
// Output: [1 2 4 1 6]
```

4. Add cards to the top of the stack

Add the card(s) specified in the `value` parameter at the top of the stack.

```
slice := []int{3, 2, 6, 4, 8}
cards := PrependItems(slice, 5, 1)
fmt.Println(cards)
// Output: [5 1 3 2 6 4 8]
```

If no argument is given for the `value` parameter, then the result equals the original slice.

```
slice := []int{3, 2, 6, 4, 8}
cards := PrependItems(slice)
fmt.Println(cards)
// Output: [3 2 6 4 8]
```

5. Remove a card from the stack

Remove the card at position `index` from the stack and return the stack. Note that this may modify the input slice which is ok.

```
cards := RemoveItem([]int{3, 2, 6, 4, 8}, 2)
fmt.Println(cards)
// Output: [3 2 4 8]
```

If the index is out of bounds (ie. if it is negative or after the end of the stack), we want to leave the stack unchanged:

```
cards := RemoveItem([]int{3, 2, 6, 4, 8}, 11)
fmt.Println(cards)
// Output: [3 2 6 4 8]
```

Source

Created by

- @tehsphinx

Contributed to by

- @norbs57